F#. Деревья
Примеры

Кузаев А.Ф.

Деревья

В дискретной математике деревом называется ациклический связанный граф.

В информатике обычно дают другое рекуррентное определение дерева общего вида типа T – ***это элемент типа T с присоединенными к нему 0 и более поддеревьями типа T.***

Если к элементу присоединено 0 поддеревьев, он называется терминальным, или **листом**, в противном случае **узлом**.

Деревья

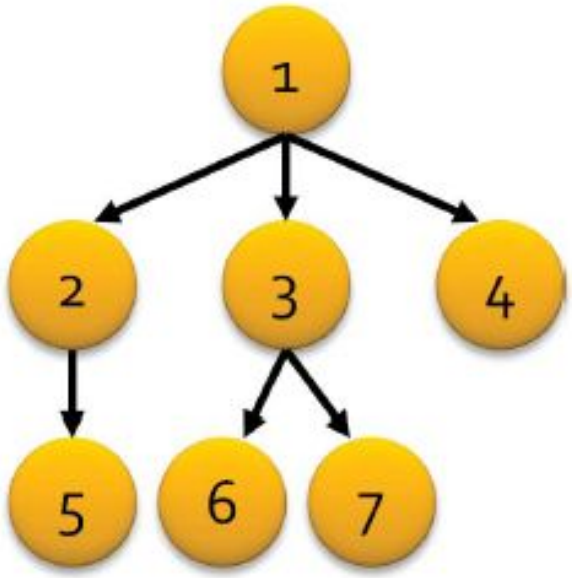
В соответствии с этим дерево может быть представлено в следующем образом:

```
type 'T tree =
    Leaf of 'T
  | Node of 'T * ('T tree list)
```

Деревья

Дерево, представленное на рис., может быть описано следующим образом:

let tr = Node(1, [Node(2, [Leaf(5)]); Node(3, [Leaf(6); Leaf(7)]); Leaf(4)])



Деревья

Основная процедура обработки дерева – это обход, когда каждый элемент дерева посещается (то есть обрабатывается) ровно 1 раз. Обход может быть с порождением другого дерева (map), или с аккумулятором (fold), но при этом базовый алгоритм обхода остается неизменным:

```
let rec iter f = function
    Leaf(T) -> f T
  | Node(T, L) -> (f T; for t in L do iter f t done)
```

Деревья

Иногда бывает полезным включать в обход также глубину соответствующего элемента, то есть количество узлов, отделяющее его от вершины:

```
let iterh f =
  let rec itr n = function
    Leaf(T) -> f n T
    | Node(T, L) -> (f n T; for t in L do itr f (n+1) t done) in
  itr 0
```

Деревья

Например, для красивой распечатки дерева с отступами можно использовать эту функцию (здесь вспомогательная функция spaces генерирует строку из *n* пробелов):

```
let spaces n = List.fold (fun s _ -> s+" ") "" [0..n]
let print_tree T = iterh (fun h x -> printf
    "%s%A\n" (spaces (h*3)) x) T
```

Пример 1

```
open System
type 'T Tree = Node of 'T * ('T Tree list)
let treemin t =
   let rec tmin min = function
      Node(m,l) ->
      let mm = if m<min then m else min
      if l=[] then mm
      else List.fold tmin mm l
   match t with
   Node(root,_) -> tmin root t

let sumt t =
   let rec tsum sum = function
      Node(m,l) ->
         let sum1 = if m>0 then sum+m else sum
         if l=[] then sum1
         else List.fold tsum sum1 l
   tsum 0 t
```

Пример 1 (продолжение)

```
[<EntryPoint>]
let main argv =
let tree = Node(-5,[
                Node(-3,[
                        Node(10,[]);
                        Node(20,[]);
                        Node(-100,[])]);
                Node(-10,[
                        Node(215,[
                                Node(10000,[]);
                                Node(-123450000,[
                                                Node(55,[])])]);
                        Node(-1000000,[])])])
  printfn "Деревья №1(мин. элемент): %d\n№2(сумма полож. элементов): %d"
     (treemin tree) (sumt tree)
  Console.ReadLine()|>ignore
  0
```

Двоичные деревья

Важной разновидностью деревьев являются **двоичные деревья** – такие деревья, у каждого узла которых есть два (возможно, пустых) поддерева – левое и правое.

Двоичные деревья не являются частным случаем деревьев общего вида, поэтому их стоит рассмотреть отдельно. Интересной особенностью двоичных деревьев также является тот факт, что любое дерево общего вида может быть представлено в виде двоичного.

Двоичные деревья

В соответствии с определением двоичных деревьев для их описания удобно использовать следующий тип:

```
type 't btree =
    Node of 't * 't btree * 't btree
    | Nil
```

Двоичные деревья

Обход двоичных деревьев, в отличие от деревьев общего вида, различается порядком обработки левого и правого поддеревьев и самого элемента в процессе обхода. Различают три основных порядка обхода, показанных в таблице, и три симметричных им обхода, при которых правое поддерево обходится раньше левого:

Порядок обхода	Название		Пример (для дерева выражения)
Корень – левое поддерево – правое поддерево	Прямой	Префиксный	+ * 1 2 3
Левое поддерево – корень – правое поддерево	Обратный	Инфиксный	1 * 2 + 3
Левое поддерево – правое поддерево – корень	Концевой	Постфиксный	1 2 * 3 +

Двоичные деревья

Опишем функцию обхода дерева, которая будет реализовывать три порядка обхода. При этом применим следующий прием: вместо того чтобы делать переключатели в коде, ограничивая возможные обходы тремя вариантами, будем описывать порядок обхода в виде функции, которая принимает три функции-аргумента для обработки корня, левого и правого поддеревьев и выполняет их в нужном порядке:

```
let prefix root left right = (root(); left(); right())
let infix root left right = (left(); root(); right())
let postfix root left right = (left(); right();root())
```

Двоичные деревья

Описав таким образом три порядка обхода, нам останется в самой процедуре обхода лишь описать три соответствующие функции для обработки корня, левого и правого поддеревьев и передать их как аргументы в переданный в виде аргумента порядок обхода trav:

```
let iterh trav f t =
  let rec tr t h =
    match t with
     Node (x,L,R) -> trav
                     (fun () -> (f x h))  // обход корня
                     (fun () -> tr L (h+1))  // обход левого поддерева
                     (fun () -> tr R (h+1)); // обход прав. поддерева
     | Nil -> ()
  tr t 0
```

Двоичные деревья

Пример инфиксного обхода дерева с использованием этой процедуры:

```
let print_tree T = iterh infix (fun x h -> printf "%s%A\n" (spaces h)x) T
```

Посмотрим также, как реализуется процедура свертки для двоичного дерева. Для простоты будем рассматривать инфиксную свертку:

Двоичные деревья

```
let fold_infix f init t =
   let rec tr t x =
     match t with
       Node (z,L,R) -> tr L (f z (tr R x))
     | Nil -> x
   tr t init
```

С помощью такой процедуры свертки можно, например, преобразовать дерево в список:

```
let tree_to_list T = fold_infix (fun x t -> x::t) [] T
```

Деревья поиска

Дерево поиска – это двоичное дерево из элементов порядкового типа, в котором для каждого узла все элементы в левом поддереве меньше данного узла, а все элементы правого поддерева – больше.

В таком дереве достаточно легко организовать поиск элемента – начиная с корня мы смотрим, меньше или больше искомый элемент корневого значения, и спускаемся в соответствующем направлении по дереву, пока либо не находим элемент, либо не доходим до листа – что означает, что элемента в дереве нет.

Деревья поиска

Добавление в дерево поиска реализуется аналогичным образом – когда мы доходим до листа и понимаем, что элемента в дереве нет, мы присоединяем его к листу в соответствующем месте (слева или справа) в зависимости от значения ключа:

```
let rec insert x t =
  match t with
    Nil -> Node(x,Nil,Nil)
  | Node(z,L,R) -> if z=x then t
                   else if x<z then Node(z,insert x L,R)
                   else Node(z,L,insert x R)
```

Деревья поиска

Для добавления целого списка элементов в дерево можем воспользоваться сверткой, где дерево выступает в роли аккумулятора:

```
let list_to_tree L = List.fold (fun t x -> insert x t) Nil L
```

В частности, можно описать сортировку списка, преобразуя список в дерево поиска и затем дерево – в список, используя ранее реализованную нами процедуру tree_to_list:

```
let tree_sort L = (list_to_tree >> tree_to_list) L
```

Пример 2

Каких чисел в дереве больше: положительных или отрицательных?

```
open System
type 't btree =                    //описание
    Node of 't * 't btree * 't btree
    | Nil
```

Пример 2

```
[<EntryPoint>]
let main A =
 let infix root left right = (left(); root(); right())  //порядок обхода
 let iterh trav f t =  //обход: здесь trav - функция порядка
    обхода
   let rec tr t h =
      match t with
      Node (x,L,R) -> trav
                      (fun () -> (f x h)) // обход корня
                      (fun () -> tr L (h+1)) // обход левого поддерева
                      (fun () -> tr R (h+1)); // обход правого поддерева
      | Nil -> ()
   tr t 0
```

Пример 2

```
let spaces n = List.fold (fun s _ -> s+" ") "" [0..n]
let print_tree T = iterh infix (fun x h -> printfn "%s%A"
    (spaces (h+3))x) T
 let rec insert x t =
  match t with
   Nil -> Node(x,Nil,Nil)
   | Node(z,L,R) -> if x<z then Node(z,insert x L,R)
                    else Node(z,L,insert x R)
```

Пример 2

```
let L =
  [
   let r = new Random()
   printfn "Количество элементов?"
   let n = Convert.ToInt32(Console.ReadLine())
   for i in 1..n do
     yield r.Next(-15, 16)
  ]
printfn "Исходный список %A" L
let list_to_tree L = List.fold (fun t x ->  insert x t) Nil L
let BT = list_to_tree L
print_tree BT
```

Пример 2

```
let fold_infix f init t =
  let rec tr t x =
   match t with
    Node (z,L,R) -> tr L (f z (tr R x))
    | Nil -> x
  tr t init
```

Пример 2

```
let solution x acc =
    if x < 0 then
        acc - 1
    else if x > 0 then
            acc + 1
         else acc
 let res x =
    if x < 0 then
        printfn "Отрицательных"
    else if x > 0 then
            printfn "Положительных"
         else printfn "Одинаковое количество"
 BT |> fold_infix solution 0 |> res
 0
```

Пример 3

В дереве содержатся натуральные числа. Построить новое дерево из тех элементов исходного, в которых есть заданная цифра *k*.

```
open System
type 't btree =                    //описание
   Node of 't * 't btree * 't btree
   | Nil
…
```

Пример 3

```
let rec Flag x c =
    if x=0 then false
    elif x%10=c then true
    else Flag (x/10) c
 printf "Искомая цифра: "
 let c = Convert.ToInt32(Console.ReadLine())
 let solution x acc =
    if Flag x c then insert x acc
    else acc
 printfn "Новое дерево: "
 BT |> fold_infix solution Nil |> print_tree
 0
```